44506

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In Application of : **KALLNER et al.**

Serial No.: 10/053,872 : Group Art Unit: 2142

Filed : January 24, 2002 : Examiner: Benjamin A. Ailes

For : COMMUNICATION ENDPOINT SUPPORTING MULTIPLE PROVIDER MODELS

Honorable Commissioner for Patents
P.O. Box 1450
Alexandria, Virginia 22313-1450

**DECLARATION UNDER 37 CFR 1.131**

Sir:

We, the undersigned, Samuel Kallner, Lev Kozakov, Alexey Roytman, Uri Shani, and Pnina Vortman, hereby declare as follows:

1) We are the Applicants in the patent application identified above, and are the inventors of the subject matter described and claimed in claims 1, 4-16, 26, 29-41, 56 and 59-71 therein.

2) Prior to March 14, 2000, we conceived our invention, as described and claimed in the subject application, in Israel, a WTO country. We then worked diligently on reducing the invention to practice (in the form of software code in the Java programming language) during a period that began prior to

March 14, 2000, and continued until the invention was actually reduced to practice and tested successfully on or about June 21, 2006. The software we created was a special implementation of the JTAPI specification, which we referred to as "Generic JTAPI" (or GenJTAPI). It provided an application programming interface (API), which enabled calls to be connected between parties via different service providers with different telephony signaling stacks, using an abstract call model, as recited in the claims of this patent application.

3) As evidence of the conception of the present invention, we attach hereto, in Exhibit A, a document that we completed on March 13, 2000, entitled "Network Infrastructure Design." The table below shows the correspondence between the elements of the method claims in the present patent application and the disclosure in Exhibit A:

| **Claim 1** | **Exhibit A** |
|---|---|
| 1. A method for communication, comprising: | The document relates to communication over "telephony networks" (line 1). |

| | |
|---|---|
| receiving a request from a first party, submitted via a first communication service provider to a telephony application, to place a call using the application to a second party; | As shown in Figure 4 (page 7), a source phone places a call using an application layer in a gateway to a destination phone. Figs. 5-7 show that calls of this sort may be placed between a PSTN phone (which uses the PSTN as a communication service provider) and an IP phone (which uses a packet network service provider). For calls originating from the PSTN phone, the PSTN is the first service provider. |
| responsive to a characteristic of the call placed by the first party, selecting a second communication service provider to carry the call between the application and the second party; and | As shown in Figure 7 and explained in detail on page 13 (steps 4-9), the gateway recognizes that the call destination (a characteristic of the call) is an IP phone and redirects the call to an IP gateway (the second service provider), which carries the call to the destination phone. |
| connecting the second party via the second communication service provider to communicate with the first party using the application, | The result of the process shown on page 13 is that the source and destination phones are connected (steps 21-26). |

| | |
|---|---|
| wherein receiving the request comprises submitting the request to the application via an application programming interface (API), which exposes a platform-independent call model to the application, and wherein connecting the second party comprises connecting the call responsive to an instruction submitted by the application to the API, and | As shown in Figure 3 (page 4), the Generic JTAPI Layer has JTAPI, JTSPI, and JTSPMI APIs. It provides a platform-independent call model to the application via the JTAPI API, through which the application submits call control instructions. |
| wherein the first and second communication service providers have respective first and second telephony signaling stacks, and wherein the call model comprises an abstract call model that is independent of the telephony signaling stacks used in placing calls to and receiving calls from the application. | In the examples shown in Figures 5-7, the PSTN and IP networks clearly have different protocol signaling stacks. (See also the different Telephony Stacks in Figure 3.) "The Generic Layer doesn't know anything about the configuration of the network below it…" (page 4, third paragraph), i.e., its call model is independent of the telephony stacks below it. |

| Claim 4 | Exhibit A |
|---|---|
| 4. A method according to claim 1, wherein receiving the request comprises passing the request from the first telephony signaling stack to the abstract call model via a service provider interface of the call model, and wherein connecting the second party comprises passing signals to the second telephony signaling stack via the service provider interface, wherein the service provider interface is independent of the telephony signaling stacks. | As shown in Figures 3 and 4, the telephony stacks for both the source and destination of the call are connected to the Generic JTAPI layer via respective Provider Plug-ins through the Java Telephony Service Provider Interface (JTSPI). It can be seen in Figure 3 that the JTSPI is independent of the different telephony stacks. |

| Claim 5 | Exhibit A |
|---|---|
| 5. A method according to claim 4, wherein passing the request from the first telephony signaling stack comprises using a first plug-in program to associate the signals in the first telephony signaling stack with corresponding elements of the service provider interface, and wherein passing the signals to the second telephony signaling stack comprises using a second plug-in program to associate the signals in the second telephony signaling stack with the corresponding elements of the service provider interface. | As noted above in reference to claim 4, and shown in Figures 3 and 4, the telephony stacks are connected to the JTSPI (service provider interface) via respective Provider Plug-in programs. |

| Claim 6 | Exhibit A |
|---|---|
| 6. A method according to claim 5, wherein selecting the second communication service provider comprises selecting the second plug-in program from among a plurality of the plug-in programs that are provided for interacting with the abstract call model. | As stated on page 4 of Exhibit A: "The Generic JTAPI layer dynamically loads Provider Plug-ins… Multiple Provider Plug-ins are supported simultaneously… Each plug-in would support its own domain…" |
| **Claim 7** | **Exhibit A** |
| 7. A method according to claim 6, wherein selecting the second plug-in program comprises passing information regarding the call to a service manager program via a service management interface of the abstract call model, wherein the service manager program processes the information to determine the characteristic, and selects the second plug-in program responsive to the characteristic from a registry of the plug-in programs. | Figure 4 shows Generic JTSMI (Java Telephony Service Management Interface) and JTSMI Plug-in components. As stated on page 4, "The JTSMI plug-in's job is to select the appropriate provider plug-in to handle a particular call leg." |

**Declaration under 37 C.F.R 1.131 by Kallner et al.**

| **Claim 8** | **Exhibit A** |
|---|---|
| 8. A method according to claim 1, wherein receiving the request comprises receiving an address of the second party to whom the call is to be placed, and wherein selecting the second communication service provider comprises parsing the address to determine the second communication service provider that should be selected. | As explained on page 5, the JTSMI plug-in uses "hints" or "address prefixes" to determine where to direct the call leg, i.e., to select the communication service provider to use in connecting to the destination of the call. The "address" may be a telephone number or an IP alias (page 9). |
| **Claim 9** | **Exhibit A** |
| 9. A method according to claim 8, wherein receiving the address comprises receiving a telephone number, and wherein parsing the address comprises identifying the second communication provider based on a portion of the telephone number. | The "address prefix" mentioned above may be a country code, i.e., a portion of a telephone number (page 5). |

| Claim 10 | Exhibit A |
|---|---|
| 10. A method according to claim 1, wherein selecting the second communication service provider comprises determining a communication protocol to be used in communicating with the second party, and choosing the second communication service provider such that the second communication service provider supports the communication protocol. | In the description of "Hybrid Network Calls" on page 6, "Appropriate JTSPI plug-ins [are] selected to support the protocols involved." |
| **Claim 11** | **Exhibit A** |
| 11. A method according to claim 10, wherein receiving the request from the first party comprises communicating with the first party via the first communication service provider using a first communication protocol, and wherein the communication protocol used in communicating with the second party comprises a second communication protocol, different from the first protocol. | Section 5 of Exhibit A (beginning on page 9) describes "hybrid calls," which are placed through the disclosed gateway between source and destination terminals on a PSTN and an H.323 network. These networks inherently use different communication protocols. |

| Claim 12 | Exhibit A |
|---|---|
| 12. A method according to claim 11, wherein one of the first and second communication protocols comprises a circuit-switched network protocol, while the other of the first and second communication protocols comprises a packet-switched network protocol. | Referring to Section 5, as noted above, the PSTN uses a circuit-switched protocol, while H.323 is a packet-switched protocol. |
| **Claim 13** | **Exhibit A** |
| 13. A method according to claim 1, wherein selecting the second communication service provider comprises specifying a selection rule, and applying the selection rule to the characteristic in order to determine the second communication service provider to be selected. | As explained above, the JTSMI plug-in chooses the service provider to carry the call to the destination. The JTSMI plug-in is driven by rules "to define the domains of the provider plug-ins in use" (page 5), i.e., to determine the domain to which the call destination belongs and thus to select the appropriate service provider. |

| Claim 14 | Exhibit A |
|---|---|
| 14. A method according to claim 13, wherein specifying the selection rule comprises specifying a temporal criterion, so that the second communication service provider is selected depending on a point in time at which the call is placed. | Although Exhibit A does not relate to temporal rules that may be implemented by the JTSMI plug-in, the Examiner indicated that Hetz (U.S. Patent 6,185,289, col. 7, lines 52-56), in combination with Smyk, would have led a person of ordinary skill in the art to specify time criteria according to which service providers should be selected in the context of claim 14. If this rationale were conceded to be correct, then it would have been obvious to the person of ordinary skill to create a JTSMI plug-in with this capability, thus implementing the method of claim 14. |

| Claim 15 | Exhibit A |
|---|---|
| 15. A method according to claim 1, wherein the telephony application comprises a teleconferencing application, and wherein connecting the second party comprises establishing a teleconference between the first and second parties. | Although Exhibit A does not specifically relate to teleconferencing applications, it does describe the use of the H.323 and H.248 protocols, which are commonly used in teleconferencing. The Examiner indicated that Smyk (col. 6, lines 30-32 and 43-46) would have led a person of ordinary skill in the art to establish a teleconference between the first and second parties in the context of claim 15. If this rationale were conceded to be correct, then it would have been obvious to the person of ordinary skill to interface a teleconferencing application with the gateway described in Exhibit A so as to implement the method of claim 15. |

| Claim 16 | Exhibit A |
|---|---|
| 16. A method according to claim 1, wherein the telephony application comprises a call center application, and wherein connecting the second party comprises establishing voice communications between a customer and a call center agent. | Although Exhibit A does not specifically relate to call center applications, the Examiner indicated that Smyk (col. 6, lines 30-32 and 43-46) would have led a person of ordinary skill in the art to establish voice communications between a customer and a call center agent in the context of claim 16. If this rationale were conceded to be correct, then it would have been obvious to the person of ordinary skill to interface a call center application with the gateway described in Exhibit A so as to implement the method of claim 16. |

4) Claims 26, 29-41, 56 and 59-71 recite apparatus and computer software products, with limitations similar to those of method claims 1 and 4-16. Based on the similarity of subject matter between the method, apparatus and software claims, it can similarly be seen that we conceived the invention recited in claims 26, 29-41, 56 and 59-71 prior to March 14, 2000.

5) In an earlier Declaration under 37 CFR 1.131, which we filed in this application on April 28, 2006, we submitted

software source code that we developed in order to reduce the present invention to practice. This earlier Declaration and the exhibits that we submitted with it (including Exhibit A, containing the software code) are incorporated herein by reference. In the earlier Declaration, we showed that this code implemented all the elements of the claims in this application or at least, in the case of certain dependent claims, rendered their implementation obvious. We began development of this software code prior to March 14, 2000, and worked diligently on its development between March 14 and June 21, 2000.

6) In a subsequent Declaration under 37 CFR 1.131, which we filed in this application on February 7, 2007, we submitted, as proof of our diligence, a version control listing for the HRL JTAPI project, showing dates on which files in the GenJTAPI program suite were updated. This subsequent Declaration and the exhibits that we submitted with it (including Exhibit B, the version control listing) are incorporated herein by reference. The version control listing shows that updates were performed regularly during the period between March 14 and June 21, 2000, as we debugged and improved our programs. With regard to specific classes listed in Exhibit A of the earlier Declaration, the version control listing contains the following information:

- CoreConnectTask.java (pages 91-92 in the listing) - added February 1, 2000; updates on March 12, April 12, April 16, May 9, and May 16, 2000.
- GenJtapiPeer.java (page 127 in the listing) - added November 9, 1999; updates on March 13, March 15, March 20, April 12, April 18, and May 9, 2000.
- ProviderPlugin.java (pages 148-149 in the listing) -

added December 2, 1999; updates on March 12 and May 16, 2000.

7) As we explained in our earlier declaration, the above-mentioned classes represent only a portion of the GenJTAPI software code. The version control listing that we submitted shows that other classes were also updated during the period between March 14 and June 21, 2000, for example:

- TapiCallpathFrame.java (pages 13-14 in the listing) - added April 12, 2000; updated June 7, 2000.
- GenCallCtlAddrDoNotDisturbEv.java (page 16 in the listing) - added November 9, 1999; updates on March 12, April 12, May 2, and May 16, 2000.
- RemoteProviderPlugin.java (page 179 in the listing) - added December 28, 1999; updated April 30, 2000; locked June 12, 2000.
- SimulatorProviderPlugin.java (pages 186-188 in the listing) - added December 28, 1999; updated March 15, March 20, March 28, March 29, April 10, April 17, April 30, and May 22, 2000; locked June 12, 2000.

8) For three of the classes that were included in Exhibit A of our earlier Declaration (JtsmiPlugin.java, MultyPluginJTSMI.java and HybridProviderPlugin.java), the version control listing in Exhibit B of our subsequent Declaration contained only dates that were later than June 21, 2000. These classes were created prior to the GenJTAPI test that was performed on June 21, but they were entered into the version control system only afterwards. For removal of doubt, we attach hereto in Exhibit B the source code listings of these classes with the dates of creation exposed. (These listings were included in Exhibit A of our earlier

declaration, but with the dates blacked out.) The dates of creation of the classes in the attached Exhibit B are follows:

- JtsmiPlugin.java (page 1) - May 31, 2000.
- MultyPluginJTSMI.java (page 2) - May 31, 2000.
- HybridProviderPlugin.java (page 10) - June 13, 2000.

9) As we explained and proved in our previous Declarations, the GenJTAPI software described above was tested in handling actual telephone traffic at the facilities of Sonera (a Finnish telecommunication service provider) on or about June 21, 2006. The successful test was described in an e-mail letter written by the project leader, Pnina Vortman, which was attached to the Declaration that we submitted on February 7, 2007, as Exhibit C, and which is incorporated herein by reference.

We hereby declare that all statements made herein of our own knowledge are true and that all statements made on information and conjecture are thought to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application of any patent issued thereon.

**Declaration under 37 C.F.R 1.131 by Kallner et al.**

Samuel Kallner
Citizen of Israel
52 Tal Menashe
D.N. Menashe 37867
Israel

Date:
4/7/2007

Lev Kozakov
Citizen of Israel
80/8 Hatishbi Street
Haifa 34523
Israel

Date:

~~Alexey Roytman~~
Citizen of Israel
68/6 Ben-Zvi Street
Kiryat Ata 28065
Israel

Date:
25.06.07

Uri Shani
Citizen of Israel
71 Givat Adi
17940
Israel

Date:
16.07.07

Pnina Vortman
Citizen of Israel
21 Netiv Ofakim
Haifa 34467
Israel

Date:
16.7.07

US 10/053,872

**Declaration under 37 C.F.R 1.131 by Kallner et al.**

Samuel Kallner
Citizen of Israel
52 Tal Menashe
D.N. Menashe 37867
Israel

Date:

Lev Kozakov
Citizen of Israel
80/8 Hatishbi Street
Haifa 34523
Israel

Date: June 26, 2007

Alexey Roytman
Citizen of Israel
68/6 Ben-Zvi Street
Kiryat Ata 28065
Israel

Date:

Uri Shani
Citizen of Israel
71 Givat Adi
17940
Israel

Date:

Pnina Vortman
Citizen of Israel
21 Netiv Ofakim
Haifa 34467
Israel

Date:

EXHIBIT A

# Network Infrastructure Design

## 1 Overview

To bring intelligence to the telephony networks in a more rapid fashion, off of the switch service nodes (SN) were added to the network. These SNs were to have made it easier to add intelligence to the network by writing services that run on the SN. There are standards for the protocols that enabled SNs to interoperate, enabling distributed services to written on different providers' platforms. However, there is no standard for the application programming interfaces (APIs) that are used to write services. Thus, if a telephony service provider wanted to deploy a service on more than one platform in his network, he had to implement that service multiple times.

With the ongoing convergence of the PSTN, wireless, and Voice over IP (VoIP) telephony networks, this problem is even more acute. As people move from one network to another, they would like to do so seamlessly, without the loss of services. Service providers would like to provide all of their services on all of their networks. This becomes difficult and expensive as every service needs to be implemented several times for each network.

A goal of the Java Service Creation project is the creation of telephony and data services which are not connected to any particular network or any particular network hardware. The goal is a layered system in which different layers can be replaced to enable the overall platform to be matched with the real underlying networks as needed. However, the API exposed to the service writer would still be completely isolated from such changes below in the platform.

Stated in a slightly different fashion, this goal is the desire to merge the PSTN, wireless (cellular), and VoIP networks into one hybrid telephony network. This goal is apparent in several ways, including the desire to:

1. Have a single service platform for all of the networks.
2. To hide the network structure from services, enabling them to be, truly, network independent.
3. Enabling calls that span the networks.

The use of an abstract telephony API, such as the Java Telephony API (JTAPI), enables the goal of a single service platform for all of the networks, as the services are in no way tied to a specific network or protocol. The underlying infrastructure takes care of the necessary translation for the underlying network.

This document describes the design of the JTAPI implementation used to merge the telephony networks into one hybrid network. Also discussed, albeit at a high level, is the physical merging of the networks.

# 2 The Generic JTAPI Implementation

As with many other Java specifications, JTAPI comes in two parts. The first part is a set of Java Interfaces and Exception classes that are standard and can be gotten from Sun Microsystems' web site. The second part is a set of classes that implements the Java Interfaces found in the first part. This separation enables many different vendors to implement the JTAPI specification, while leaving applications independent of (and probably not even knowing) the names of the classes in any particular JTAPI implementation.

In general a JTAPI implementation looks like the picture in Figure 1 below. As described above, the application above the JTAPI API is completely independent of the JTAPI implementation below the JTAPI API.

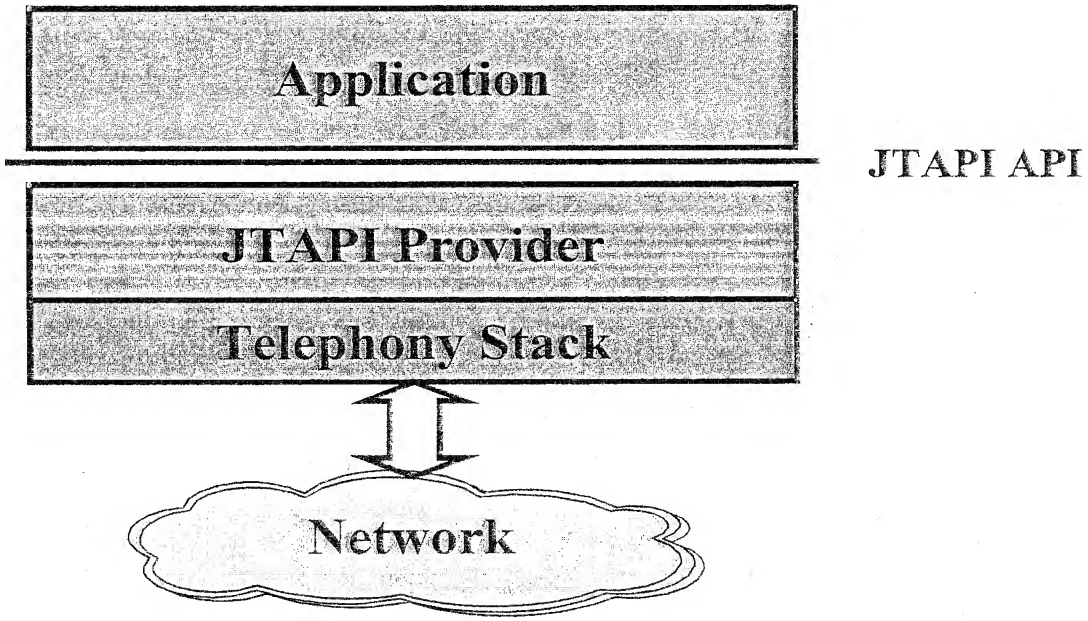


**Figure 1. A typical JTAPI implementation.**

While a JTAPI application is independent of the JTAPI implementation below it, a JTAPI implementation is typically tightly bound with the telephony stack below it. Portions of the JTAPI implementation need to map between JTAPI objects and methods and the telephony stack's interfaces.

Large portions of a JTAPI implementation are actually independent of the telephony stack used. Code that checks pre-conditions, deals with events, and does the base object manipulation is completely independent of the telephony stack. However, telephony stack

specific code needs to be sprinkled thru out this common code. Thus making it harder to share code between implementations.
The IBM Haifa Research Lab's Generic JTAPI Implementation was designed to overcome this specific problem, providing the following benefits:

1. Enabling JTAPI implementations for various telephony stacks and platforms to be written quickly.
2. The JTAPI specification is subject to much interpretation. All implementations based on the Generic JTAPI Implementation would have been written with a single interpretation.
3. Lower resource usage in cases where multiple telephony stacks are used simultaneously.

In the Generic JTAPI Implementation the JTAPI Provider is split into two layers:

1. The common JTAPI functions are in the Generic JTAPI layer.
2. The telephony stack specific function is in the Provider Plug-in layer.

As can be seen in Figure 2 below, the generic layer interfaces with the plug-in layer via the Java Telephony Service Provider Interface (JTSPI).

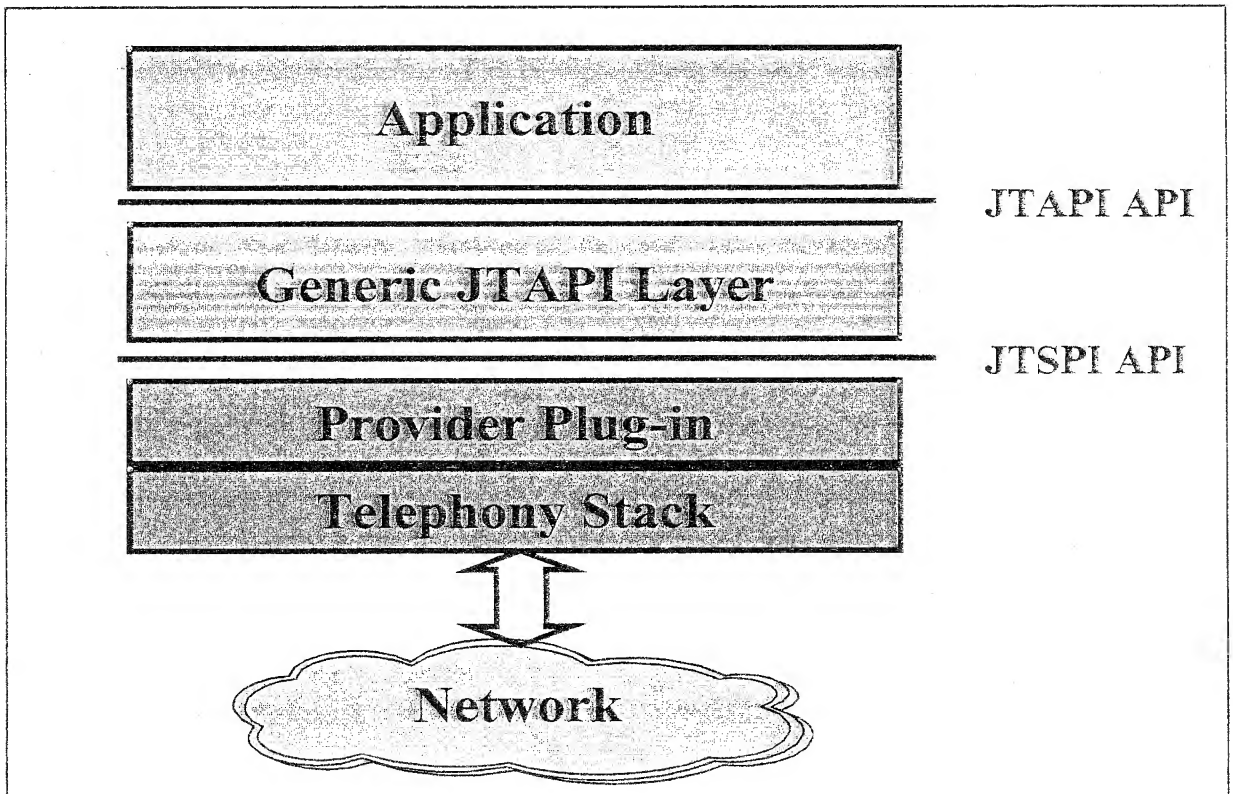


**Figure 2.Generic JTAPI Implementation**

# Network Infrastructure Design

The Generic JTAPI layer dynamically loads Provider Plug-ins with the use of configuration data. Thus a "custom" JTAPI implementation tailored to the specific telephony stack being used can be built with ease.

Multiple Provider Plug-ins are supported simultaneously. In such cases they are assumed to be connected to different telephony networks thru the appropriate telephony stacks. Each plug-in would support it's own domain made up of a set of local and remote addresses as appropriate. However, from an application perspective, these networks can be viewed as one hybrid network. Thus meeting the project goal of a single telephony API over multiple networks on the same platform.

As the Generic Layer doesn't know anything about the configuration of the network below it, another component was added to the IBM Haifa JTAPI Implementation. This component is the Java Telephony Service Management Interface (JTSMI) plug-in.

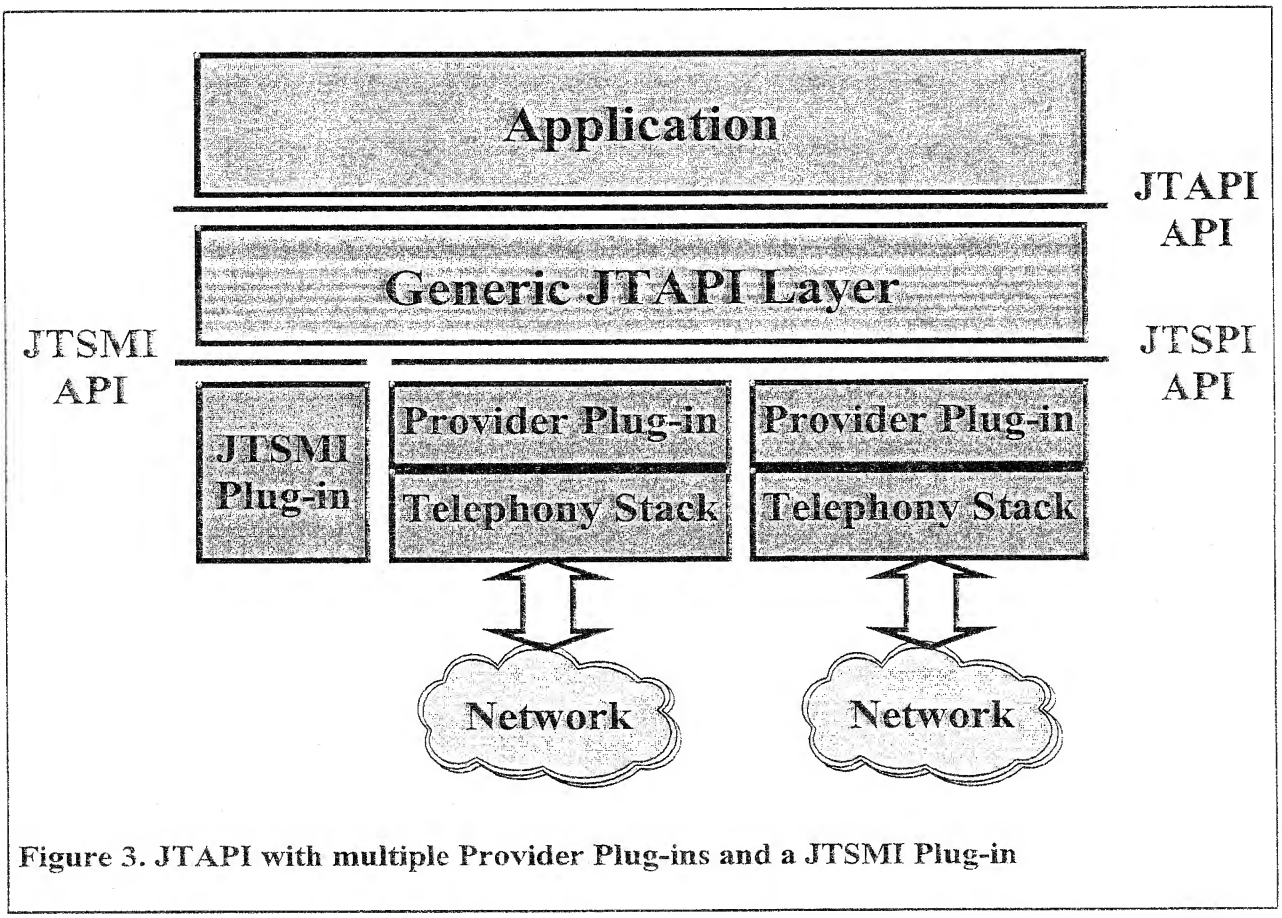


**Figure 3. JTAPI with multiple Provider Plug-ins and a JTSMI Plug-in**

As can be seen in Figure 3,above, the Generic JTAPI layer interfaces with the JTSMI plug-in thru the JTSMI API. The JTSMI plug-in's job is to select the appropriate provider plug-in to handle a particular call leg. Calls to the provider plug-in are actually made thru

the JTSMI plug-in. In fact, in some cases the JTSMI plug-in may issue different calls to the plug-ins than were made to it by the Generic JTAPI Layer. As such, the JTSMI plug-in is often tailored to the environment and understands the addressing conventions used in the applications running on top. It is planned that an external rules driven JTSMI plug-in will be written to avoid the need to constantly write new JTSMI plug-ins. For such a JTSMI plug-in new rules will be written to define the domains' of the Provider plug-ins in use.

The JTSMI plug-in also has knowledge of, at least at a high level, of the configuration of the network. Depending on the underlying network configuration, "hints" or "address prefixes" may be needed to tell the JTSMI plug-in where to direct the call leg.[1] Thought should be given to the use of the newly allocated ITU country code(s) for VoIP where appropriate.

[1] This is especially true if PSTN calls are being routed over VoIP networks in a Long Distance Bypass type of scenario.

# 3 Hybrid Network Calls

One of the main issues in determining exactly how calls that span networks work is the capabilities of the Gateway (GW) involved.

In general when bridging two networks both a Signaling GW and a Media GW are needed. The former translates signaling messages, while the later transcodes the media stream. While fundamentally there are two GWs involved in a call, often in fact, there is one GW that has combined the Signaling GW and the Media GW in one box. Most GWs built in this fashion expect to handle both the Signaling Translation and Media Transcoding. Efforts such as the IETF's MGCP and the ITU's Gateway Decomposition work are geared towards providing standard protocols to integrate decomposed GW. Such a decomposed GW is usually made up of:

1. A control element, called the Gateway Controller (GC).
2. A Signaling Gateway (SG) to translate signaling messages.
3. A Media Gateway (MG) to transcode the media streams.

Figure 4 on page 7 shows how a GC and a SG can be built from:

1. IBM HRL's Generic JTAPI.
2. Appropriate JTSPI plug-ins selected to support the protocols involved.
3. An appropriate JTSMI plug-in selected/written to manage the JTSPI plug-ins and the MG selected.
4. An application that redirects calls from one network's addresses to the other network's addresses[2].

[2] The application need not realize that it is in fact redirecting calls between the networks. There does need to be some sort of address translation/replacement to direct the JTSMI to create a hybrid network call.

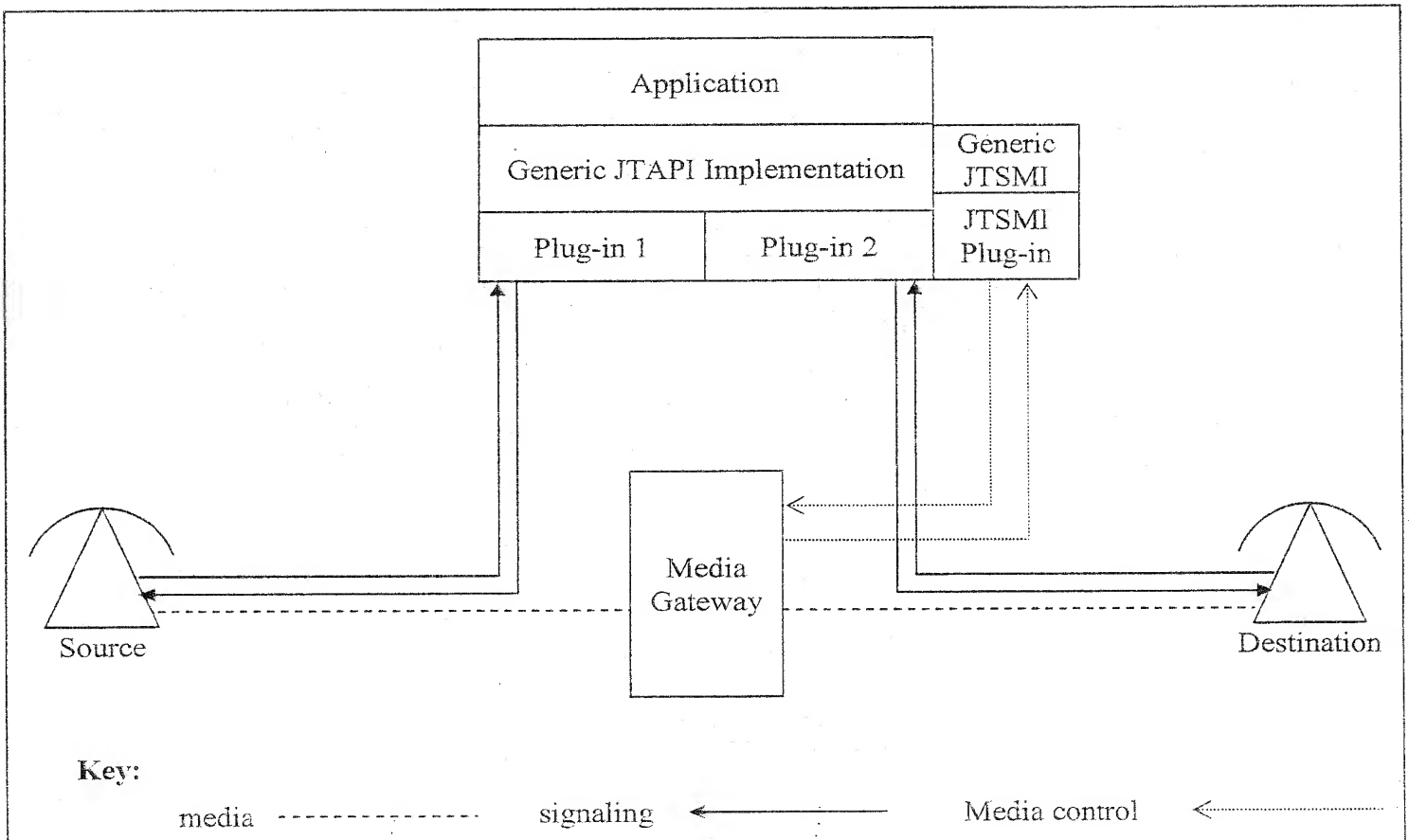


**Figure 4 An example of a decomposed Gateway**

In Figure 4, above, a third party calling model is shown. That is, a third party calling model from the perspective of the JTAPI application. However, from the perspective of the two JTSPI plug-ins involved, a pair of first party calls are placed. One call is an incoming call to Plug-in 1. The other call is an outgoing call from Plug-in 2. The Generic JTAPI combined with the JTSMI plug-in will make it look to the second plug-in as if the "second call" was originated by the real source of the hybrid call.

# 4 The Service Node

Services Nodes (SN) are made up of three main components:

1. A Service Execution Environment (SEE) component in which the services run.
2. A management component, which manages the SN.
3. A telephony network infrastructure component, which connects the SEE to the telephony networks below.

The design of the SEE and SN management component are discussed in their respective design documents.

The telephony network infrastructure component used in the Service Creation Project is IBM Haifa Research Lab's Generic JTAPI Implementation discussed in the section "The Generic JTAPI Implementation" on page 2.

From the perspective of the JTAPI implementation inside the SN the rest of the SEE is simply a JTAPI application. That is to say that it knows nothing special about the SEE.[3]

The exception to this is the JTSMI plug-in, which is in charge of associating call legs with the appropriate Provider plug-in. The JTSMI plug-in will need to understand the structure and addressing conventions of the networks below the JTAPI implementation. The use of an appropriate network-naming scheme along with an appropriate JTAPI implementation also enables the hiding of the underlying network's structure.

In particular, the JTSMI will need to handle PSTN (including both wired and wireless) and H.323 VoIP calls.

[3] It should be noted that in a SN with high availability characteristics there might be a need to have very tight integration between the rest of the SN and the JTAPI implementation. This is particularly true for the saving and restoring of the call state.

# 5 The Sonera Test Network

In the Sonera Test Network we will not have a decomposed GW. The RadVision GW is a monolithic GW that includes an SG and an MG under the covers. On the PSTN side it supports DTMF digit input using normal PSTN signaling. Calls can be routed thru it from PSTN to the H.323 network using single stage dialing. The GateKeeper (GK) will route H.323 calls that originate from the GW.

Based on this, all hybrid network calls originating from the PSTN must be routed to the H.323 network via a phone number. However, the H.323 terminal being called need not have registered itself with the GK with an E.164 alias. As a result this "phone number" must be:

1. Recognized by the PSTN and routed to the GW
2. Recognized by the GK and routed to the correct IP Terminal.

Our proposal is to make use of one of the new ITU-T E.164 country codes reserved for IP Telephony. All phone numbers with a specific country code and area code could be routed by the PSTN to our test network's GW. The rest of the phone number could either be:

1. The IP Terminal's IP address and port, represented numerically perhaps in the form aaabbbccddddpppp.
2. A "cookie" used by the GK to recognize the desired IP alias to be translated.

In either case, neither the service nor the Generic JTAPI layer need know that such a "translation" is occurring. One of the JTSMI's jobs is to direct all calls from the Generic JTAPI layer to the correct JTSPI plug-in. When it does this it is capable of "translating" parameters passed. When it sees an IP alias used as a party address of a call that originated from the PSTN, it can replace the alias with a "phone number". The JTSMI plug-in could either:

1. Access the GK's Registration, Access, and Status (RAS) Data Base (DB) directly or via some side protocol.
2. Somehow register a numeric cookie for later use. The cookie would need to be destroyed after use as well.

In either case the GK's call routing code would need to recognize the special "phone number" and either use the IP address and port sent or lookup the signaling address via the specified cookie.

Figure 5 "Hybrid PSTN to H.323 call." on page 10, depicts a hybrid call originating in the PSTN and redirected to the H.323 network. A more detailed call flow can be seen in the section "Hybrid PSTN to H.323 Call Flow" on page 12.

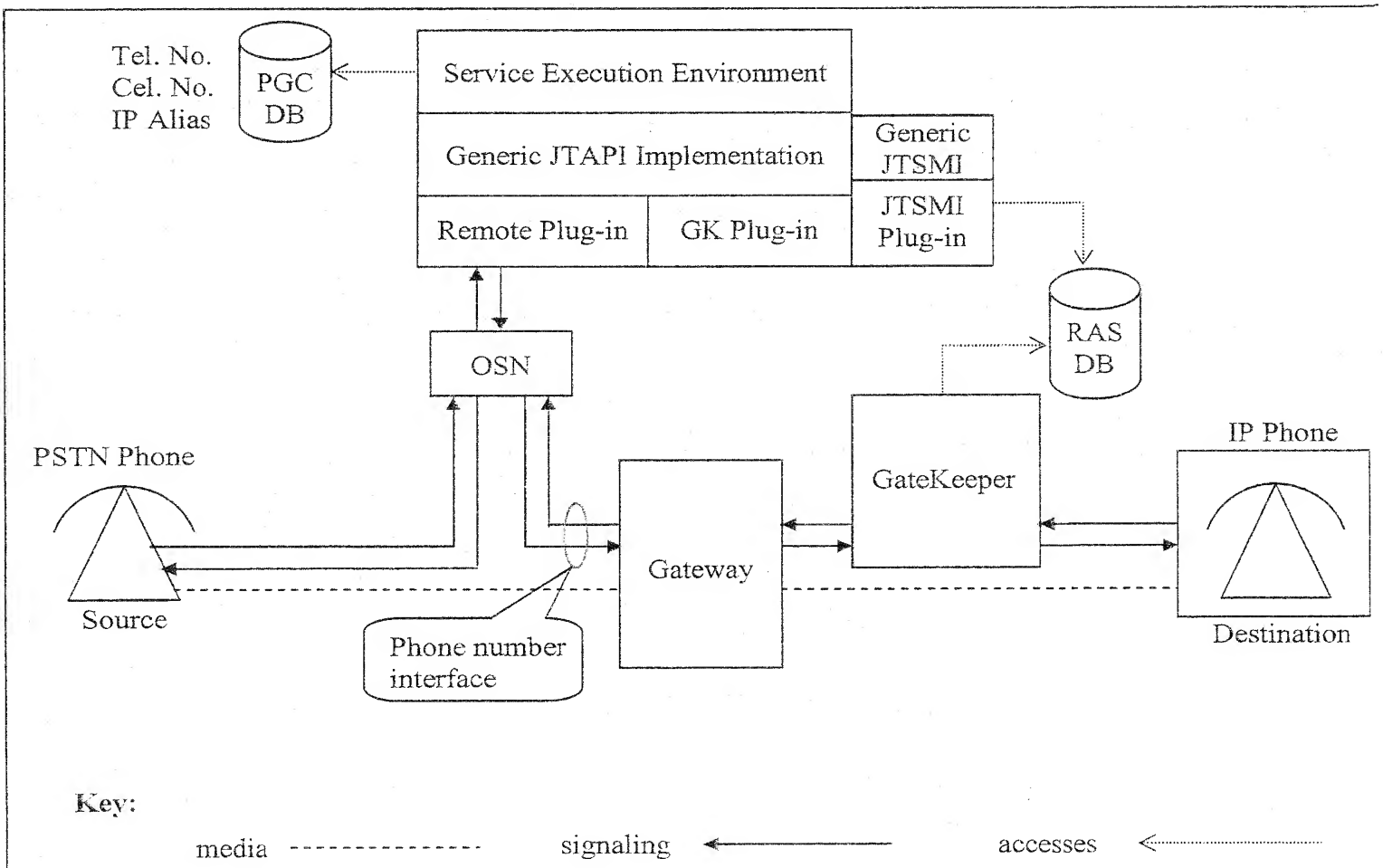


**Figure 5. Hybrid PSTN to H.323 call.**

While a hybrid PSTN to H.323 call is rather involved, a hybrid H.323 to PSTN all is much simpler. As the H.323 protocol supports connecting to an E.164 format phone number the call can be simply routed by the GK through the GW to the PSTN.

Figure 6 "Hybrid H.323 to PSTN call." on page 11, depicts a hybrid call that originated in the H.323 network and was redirected to the PSTN. A more detailed call flow can be seen in the section "Hybrid H.323 to PSTN Call Flow" on page 14.

# Network Infrastructure Design

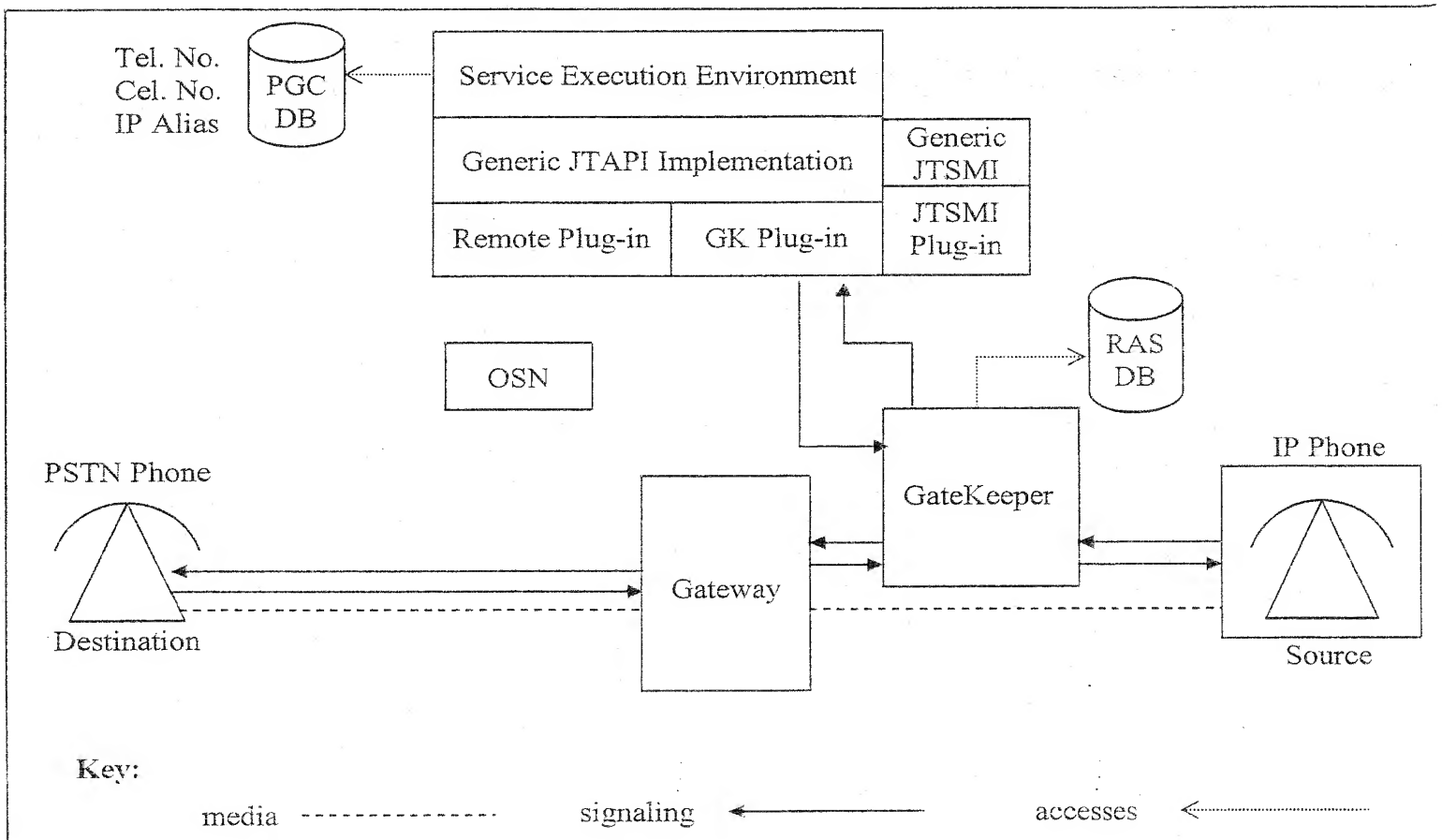


**Figure 6. Hybrid H.323 to PSTN call.**

# 6 Call Flows

## 6.1 Hybrid PSTN to H.323 Call Flow

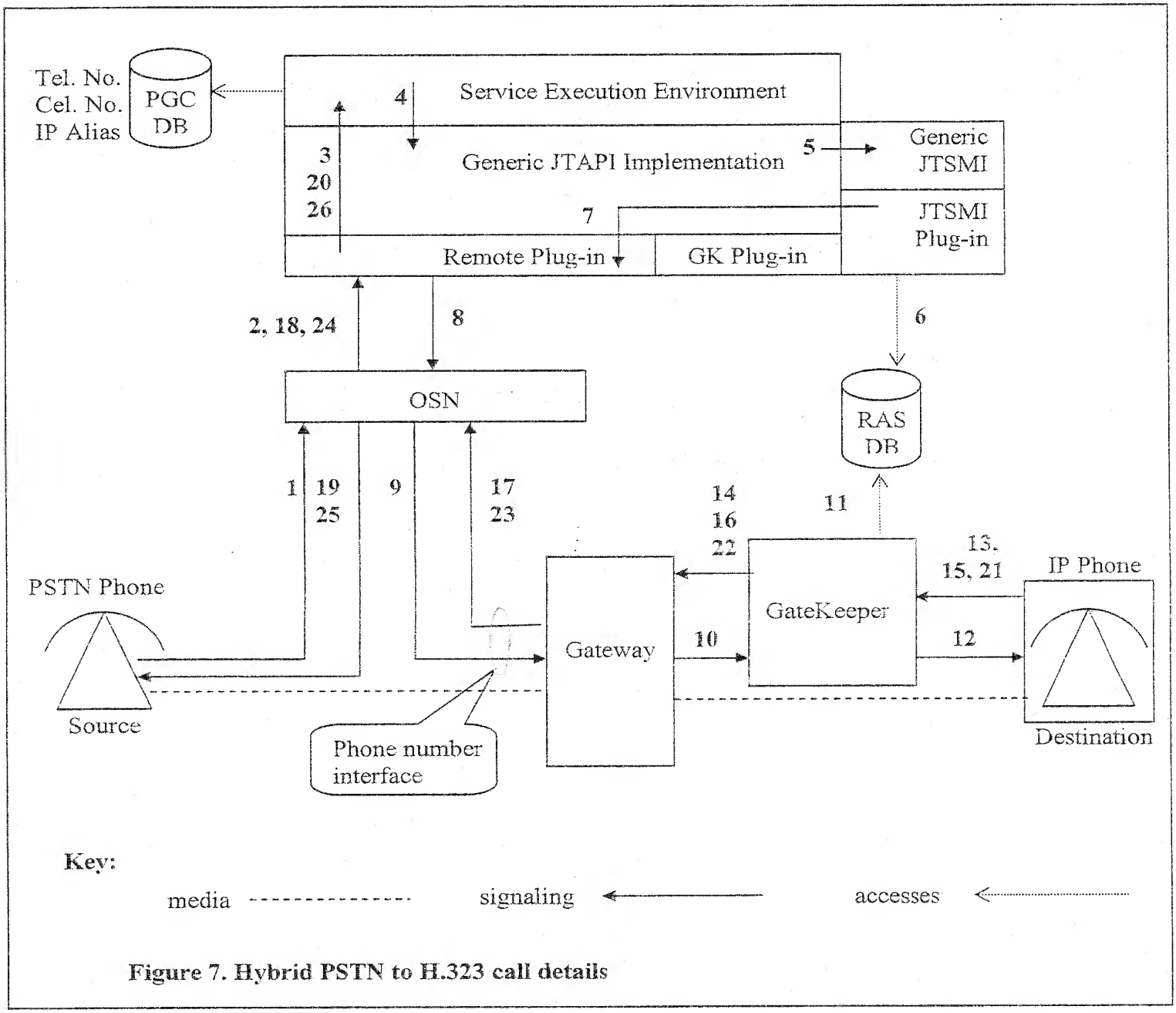


**Figure 7. Hybrid PSTN to H.323 call details**

### 6.1.1 Hybrid PSTN to H.323 Message Flow

**Note:** In the following description, SS7 messages are depicted by their three letter abbreviations in upper case (e.g. IAM). Messages of the Remote JTSPI protocol are depicted in italics (e.g. *Offering*). H.323 messages are depicted in all upper case (e.g. SETUP).

1. A call is placed on the PSTN and an IAM message is sent by PSTN to the Open Service Node (OSN).
2. An *Offering* message sent by the OSN to the Remote JTSPI Plug-in.
3. A CallCtlOfferedEv is sent to the SEE.
4. The service calls the redirect() method of the connection specifying an IP alias as the destination address.
5. The Generic JTAPI calls a method of the JTSMI to have the "redirect()" method of the Remote plug-in invoked.
6. The JTSMI plug-in recognizes a hybrid call that originates in the PSTN. It either looks up the IP alias in the RAS DB or registers a numeric cookie.
7. The "redirect()" method of the Remote JTSPI Plug-in is invoked with the special numeric destination address.
8. The *Redirect* message is sent to the OSN, again with the special numeric destination address.
9. The call is redirected by the PSTN to the GW.
10. The GW sends a SETUP message to the GK with a destination alias of the special numeric phone number.
11. The GK receives the SETUP message and, if necessary, looks up the cookie and translates the IP alias to the actual signaling address.
12. The GK sends the SETUP message to the destination IP Phone.
13. The IP Phone sends a PROCEEDING message to the GK.
14. The GK forwards the PROCEEDING message to the GW.
15. The IP Phone starts to ring and sends an ALERTING message to the GK.
16. The GK forwards the ALERTING message to the GW
17. The GW sends the OSN an ACM message
18. The OSN sends a *Ringing* message to the Remote JTSPI Plug-in.
19. The OSN forwards the ACM message to the PSTN.
20. CallCtlConnAlertingEv and CallCtlTermConnRingingEv events are sent to the SEE.
21. The IP Phone is answered and sends a CONNECTED message to the GK.
22. The GK forwards the CONNECTED message to the GW.
23. The GW sends an ANM message to the OSN.
24. The OSN sends a *Connected* message to the Remote JTSPI Plug-in.
25. The OSN forwards the ANM message to the PSTN.
26. CallCtlConnEstablishedEv and CallCtlTermConnTalkingEv events are sent to the SEE.

## 6.2 Hybrid H.323 to PSTN Call Flow

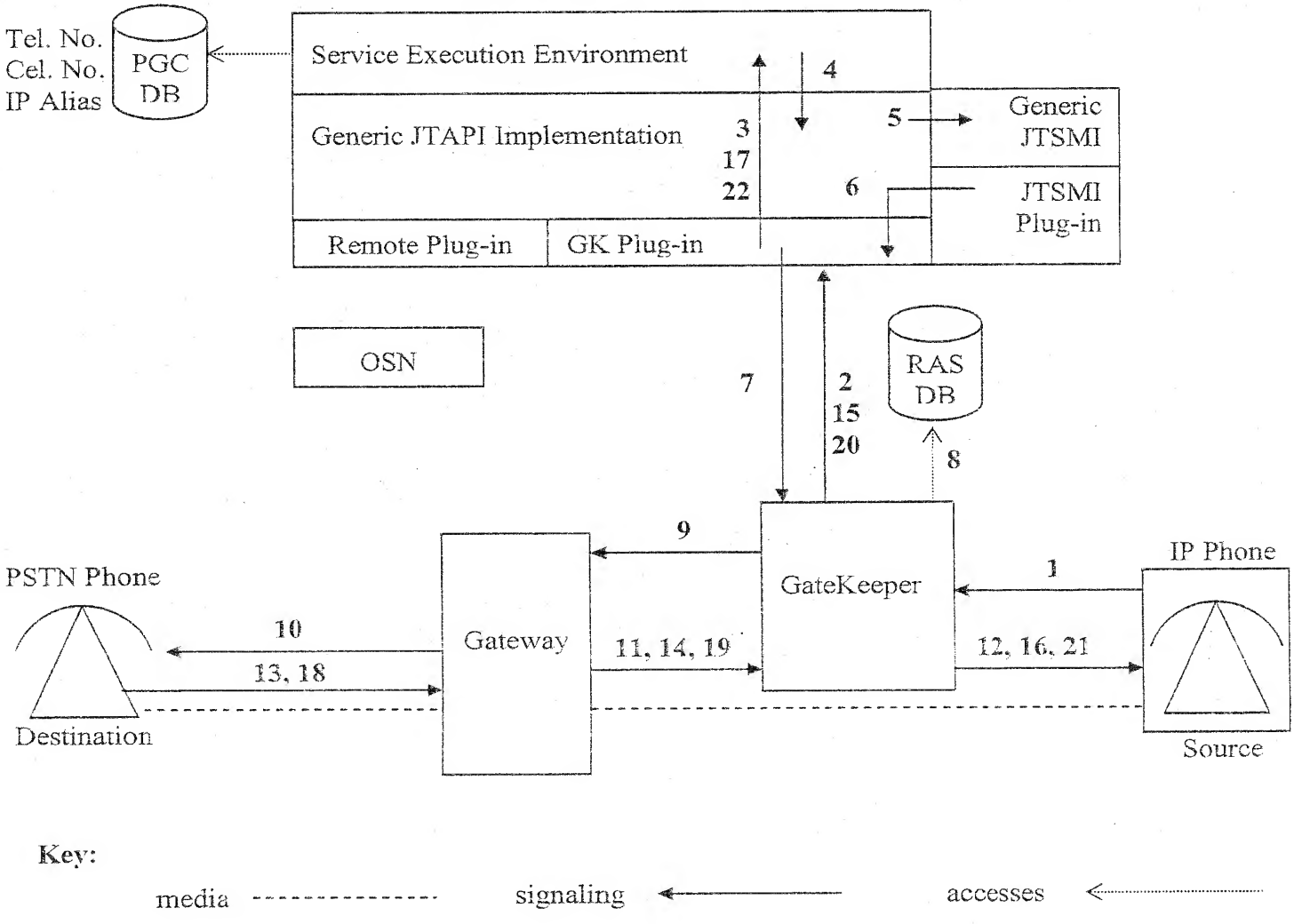


**Figure 8. Hybrid H.323 to PSTN Call Details**

### 6.2.1 Hybrid H.323 to PSTN Message Flow

**Note:** In the following description, SS7 messages are depicted by their three letter abbreviations in upper case (e.g. IAM). Messages of the Remote JTSPI protocol are depicted in italics (e.g. *Offering*). H.323 messages are depicted in all upper case (e.g. SETUP).

1. A call is placed on an IP Phone and a SETUP message is to the GK.
2. An *Offering* message sent by the GK to the GK's Remote JTSPI Plug-in.
3. A CallCtlOfferedEv is sent to the SEE.
4. The service calls the redirect() method of the connection specifying an PSTN phone number as the destination address.
5. The Generic JTAPI calls a method of the JTSMI to have the "redirect()" method of the Remote plug-in invoked.
6. The JTSMI plug-in recognizes a call that originates in the H.323 network and invokes the "redirect()" method call of the GK's Remote JTSPI Plug-in.
7. The *Redirect* message is sent to the GK, again with the address.
8. The GK looks up, if necessary the signaling address of the specified E.164 alias.
9. The GK sends a SETUP message to the GW with a destination address of the specified phone number.
10. The GW sends an IAM message to the PSTN.
11. The GW sends a PROCEEDING message to the GK.
12. The GK forwards the PROCEEDING message to the IP Phone.
13. The destination phone starts to ring and the PSTN sends an ACM message to the GW.
14. The GW sends an ALERTING message to the GK
15. The GK sends a *Ringing* message to the Remote JTSPI Plug-in
16. The GK forwards the ALERTING message to the IP Phone.
17. CallCtlConnAlertingEv and CallCtlTermConnRingingEv events are sent to the SEE.
18. The destination phone is answered and the PSTN sends an ANM message to the GW.
19. The GW sends a CONNECTED message to the GK.
20. The GK sends a *Connected* message to the Remote JTSPI Plug-in.
21. The GK forwards the CONNECTED message to the IP Phone.
22. CallCtlConnEstablishedEv and CallCtlTermConnTalkingEv events are sent to the SEE.

# 7 Issues

## 7.1 Future Work

1. Potential work, not within the scope of this project, would be to write H.248/MEGACO JTSPI and JTSMI plug-ins which could control external SG's and MG's. In such a configuration the H.248/MEGACO JTSPI and JTSPI plug-ins, the Generic JTAPI, and an application would simply be a GC.

**EXHIBIT B**

```
package com.ibm.hrl.jtapi.jtsmi;

/*
 * (c) Copyright IBM Corporation 1998,1999,2000
 * IBM Research Laboratory in Haifa
 * Generic JTAPI Implementation (JTAPI 1.3)
 * ----------------------------------------
 * Package    : com.ibm.hrl.jtapi.jtsmi
 * Class      : JtsmiPlugin
 * Created    : 31.05.2000
 */

import java.util.Properties;
import java.util.Hashtable;

public interface JtsmiPlugin
{
    public void init( String ruleFile, Properties initData );

    public String addressConvert( String address );

    public String pluginForAddress( String address );

    public boolean isInService( Hashtable providerPluginStates );

}
```

```
package com.ibm.hrl.jtapi.jtsmi;

/*
 * (c) Copyright IBM Corporation 1998,1999,2000
 * IBM Research Laboratory in Haifa
 * Generic JTAPI Implementation (JTAPI 1.3)
 * ----------------------------------------
 * Package    : com.ibm.hrl.jtapi.jtsmi
 * Class      : MultyPluginJTSMI
 * Created    : 31.05.2000
 */

import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Properties;
import java.util.Vector;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import javax.telephony.Address;
import javax.telephony.Provider;
import javax.telephony.Terminal;
import javax.telephony.Call;
import javax.telephony.ResourceUnavailableException;
import javax.telephony.InvalidArgumentException;
import javax.telephony.PlatformException;

import com.ibm.hrl.jtapi.GenAddress;
import com.ibm.hrl.jtapi.GenTerminal;
import com.ibm.hrl.jtapi.GenProvider;
import com.ibm.hrl.jtapi.GenJtapiPeer;
import com.ibm.hrl.jtapi.GenConnection;

import com.ibm.hrl.jtapi.capabilities.GenAddressCapabilities;
import com.ibm.hrl.jtapi.capabilities.GenCallCapabilities;
import com.ibm.hrl.jtapi.capabilities.GenConnectionCapabilities;
import com.ibm.hrl.jtapi.capabilities.GenProviderCapabilities;
import com.ibm.hrl.jtapi.capabilities.GenTerminalCapabilities;
import com.ibm.hrl.jtapi.capabilities.GenTerminalConnectionCapabilities;

import com.ibm.hrl.jtapi.util.JtapiObjectCreator;

import com.ibm.hrl.jtapi.jtspi.JtapiCallbacks;
import com.ibm.hrl.jtapi.jtspi.ProviderPlugin;

public class MultyPluginJTSMI extends Jtsmi
{
    // IBM Copyright
      public static final String IBM_Copyright    = Copyright.SHORT_STRING;

      public static String ms_jtsmiName  = "JtsmiPlugin";

      // providers' plugins objects
      // key - plugin name
      // value - plugin object
      private Hashtable m_plugins;
```

```
        // plugin's states
        // key plugin object
        // value Boolean state
        private Hashtable m_pluginStates;


        // key - addressName
        // valuse terminal name array.
        private Hashtable m_resources;

        private JtsmiPlugin m_jtsmiPlugin;

        private static Boolean ms_inService = new Boolean( true );
        private static Boolean ms_outOfService = new Boolean( false );


        public MultyPluginJTSMI( Properties pluginsProp, Properties initData )
        {
            m_resources = new Hashtable();
            m_plugins = new Hashtable(3);
            m_pluginStates = new Hashtable(3);
            doIt( pluginsProp, initData );
        }

        private synchronized void doIt( Properties pluginsProp, final Properties initData
)
        {
            Enumeration pluginsEnum = pluginsProp.keys();
          int runningThreads = 0;
          final Vector plugins = new Vector();
          while(pluginsEnum.hasMoreElements())
          {
              final String plugName = (String)pluginsEnum.nextElement();
              String plugData = pluginsProp.getProperty(plugName);
              int delimIndex = plugData.indexOf(ms_delimeter);
              if( 1 > delimIndex || delimIndex == (plugData.length() - 1))
              {
                  continue;
              }
              final ProviderPlugin plugin;
              final String resourceFile;
              try
              {
                  Class pluginClass = Class.forName(plugData.substring(0, delimIndex));
                  if( plugName.equals(ms_jtsmiName))
                  {
                      m_jtsmiPlugin = (JtsmiPlugin)pluginClass.newInstance();
                      m_jtsmiPlugin.init( plugData.substring(delimIndex +1), initData );
                      continue;
                  }
                  plugin = (ProviderPlugin)pluginClass.newInstance();
                  resourceFile = plugData.substring(delimIndex +1);
              }
              catch(Exception e)
              {
                  GenJtapiPeer.ms_log.warningMessage("GenJtapiPeer: initPlugin - " +
plugName, e);
                  // We don't use the plugin
                  continue;
```

```
            }
            runningThreads ++;
            Runnable r = new Runnable()
            {
                public void run()
                {
                    Hashtable table;
                    try
                    {
                        table = initPlugin( plugin, resourceFile, initData );
                        m_resources.put( plugin, table );
                        m_plugins.put( plugName, plugin );
                        m_pluginStates.put( plugin, ms_inService );
                    }
                    catch ( Exception e )
                    {
                        // We don't use the plugin
                        GenJtapiPeer.ms_log.warningMessage("GenJtapiPeer: initPlugin -
" + plugName, e);
                        m_resources.put( plugin, new Hashtable());

                    }
                    synchronized( MultyPluginJTSMI.this )
                    {
                        MultyPluginJTSMI.this.notify();
                    }
                }
            };
            GenJtapiPeer.runIt(r);
        }
        while( runningThreads != m_resources.size() )
        {
            try {
                wait();
            }
            catch (InterruptedException e )
            {}
        }
        int level = -1; // 111111111111....
        pluginsEnum = m_plugins.elements();
        while( pluginsEnum.hasMoreElements() )
        {
            level &= ((ProviderPlugin)pluginsEnum.nextElement()).prvSupportedLevel();
        }
        m_creator = new JtapiObjectCreator( level );
    }

    public void init( GenProvider provider) throws InvalidArgumentException
    {
        m_provider = provider;
        // Need update callback creator.
        JtapiCallbacks callbacks =
                m_creator.createJtapiCallbacks(provider, this);
        Enumeration plugins = m_plugins.elements();
        ProviderPlugin plugin;
        while( plugins.hasMoreElements() )
        {
            plugin = (ProviderPlugin)plugins.nextElement();
            plugin.prvSetCallback(callbacks);
```

```
            createEndPoints( plugin, provider, (Hashtable)m_resources.get( plugin ));

        }
        m_resources = null;
        changeProviderState();
    }


public void updateAddressCapabilities(
                                GenAddressCapabilities capabilities)
{
    GenAddressCapabilities tmpCapabilities = new GenAddressCapabilities();
    Enumeration plugins = m_plugins.elements();
    ProviderPlugin plugin;
    if( plugins.hasMoreElements() )
    {
        plugin = (ProviderPlugin)plugins.nextElement();
        plugin.updateAddressCapabilities( capabilities );
    }
    while( plugins.hasMoreElements() )
    {
        plugin = (ProviderPlugin)plugins.nextElement();
        plugin.updateAddressCapabilities( tmpCapabilities );
        capabilities.and( tmpCapabilities );
        tmpCapabilities.reset();
    }
}

/**
 * Updates <code>static</code> <code>GenTerminalCapabilities</code> for this
 * provider plugin. The <code>GenTerminalCapabilities</code> implements the
 * capability interfaces for the <code>Terminal</code> object from the JTAPI
 * core and all extension packages. By default all methods of instance of the
 * class, except <code>isObservable()</code>, return <code>false</code>.
 * By this method the plugin can change the default benaved  of the instance.
 *
 */
public void updateTerminalCapabilities(
                               GenTerminalCapabilities capabilities)
{
    GenTerminalCapabilities tmpCapabilities = new GenTerminalCapabilities();
    Enumeration plugins = m_plugins.elements();
    ProviderPlugin plugin;
    if( plugins.hasMoreElements() )
    {
        plugin = (ProviderPlugin)plugins.nextElement();
        plugin.updateTerminalCapabilities( capabilities );
    }
    while( plugins.hasMoreElements() )
    {
        plugin = (ProviderPlugin)plugins.nextElement();
        plugin.updateTerminalCapabilities( tmpCapabilities );
        capabilities.and( tmpCapabilities );
        tmpCapabilities.reset();
    }
}

/**
 * Updates <code>static</code> <code>GenTerminalConnectionCapabilities</code>
```

```
     * for this provider plugin.
     * The <code>GenTerminalConnectionCapabilities</code> implements the
     * capability interfaces for the <code>TerminalConnection</code> object from
     * the JTAPI core and all extension packages. By default all methods of
     * instance of the class, except <code>canAnswer()()</code>, return
     * <code>false</code>. By this method the plugin can change the default
     * behaved  of the instance.
     *
     */
    public void updateTerminalConnectionCapabilities(
                        GenTerminalConnectionCapabilities capabilities)
    {
        GenTerminalConnectionCapabilities tmpCapabilities =
                                            new GenTerminalConnectionCapabilities();
        Enumeration plugins = m_plugins.elements();
        ProviderPlugin plugin;
        if( plugins.hasMoreElements() )
        {
            plugin = (ProviderPlugin)plugins.nextElement();
            plugin.updateTerminalConnectionCapabilities( capabilities );
        }
        while( plugins.hasMoreElements() )
        {
            plugin = (ProviderPlugin)plugins.nextElement();
            plugin.updateTerminalConnectionCapabilities( tmpCapabilities );
            capabilities.and( tmpCapabilities );
            tmpCapabilities.reset();
        }
    }

    /**
     * Updates <code>static</code> <code>GenConnectionCapabilities</code> for
     * this provider plugin. The <code>GenConnectionCapabilities</code>
     * implements the capability interfaces for the <code>Connection</code>
     * object from the JTAPI core and all extension packages. By default all
     * methods of instance of the class, except <code>canDisconnect()()</code>,
     * return <code>false</code>. By this method the plugin can change the
     * default behaved  of the instance.
     *
     */
    public void updateConnectionCapabilities(
                                            GenConnectionCapabilities capabilities)
    {
        GenConnectionCapabilities tmpCapabilities =
                                            new GenConnectionCapabilities();
        Enumeration plugins = m_plugins.elements();
        ProviderPlugin plugin;
        if( plugins.hasMoreElements() )
        {
            plugin = (ProviderPlugin)plugins.nextElement();
            plugin.updateConnectionCapabilities( capabilities );
        }
        while( plugins.hasMoreElements() )
        {
            plugin = (ProviderPlugin)plugins.nextElement();
            plugin.updateConnectionCapabilities( tmpCapabilities );
            capabilities.and( tmpCapabilities );
            tmpCapabilities.reset();
        }
```

```
}

/**
 * Updates <code>static</code> <code>GenACallCapabilities</code> for this
 * provider plugin. The <code>GenCallCapabilities</code> implements the
 * capability interfaces for the <code>Call</code> object from the JTAPI
 * core and all extension packages. By default all methods of instance of the
 * class, except <code>isObservable()</code> and <code>canConnect()</code>,
 * return <code>false</code>. By this method the plugin can change the
 * default behaved  of the instance.
 *
 */
public void updateCallCapabilities( GenCallCapabilities capabilities)
{
    GenCallCapabilities tmpCapabilities =
                                        new GenCallCapabilities();
    Enumeration plugins = m_plugins.elements();
    ProviderPlugin plugin;
    if( plugins.hasMoreElements() )
    {
        plugin = (ProviderPlugin)plugins.nextElement();
        plugin.updateCallCapabilities( capabilities );
    }
    while( plugins.hasMoreElements() )
    {
        plugin = (ProviderPlugin)plugins.nextElement();
        plugin.updateCallCapabilities( tmpCapabilities );
        capabilities.and( tmpCapabilities );
        tmpCapabilities.reset();
    }
}

/**
 * Updates <code>static</code> <code>GenProviderCapabilities</code> for this
 * provider plugin. The <code>GenProviderCapabilities</code> implements the
 * capability interfaces for the <code>Providers</code> object from the JTAPI
 * core and all extension packages. By default all methods of instance of the
 * class, except <code>isObservable()</code>, return <code>false</code>.
 * By this method the plugin can change the default behaved  of the instance.
 *
 */
public void updateProviderCapabilities( GenProviderCapabilities capabilities)
{
    GenProviderCapabilities tmpCapabilities =
                                        new GenProviderCapabilities();
    Enumeration plugins = m_plugins.elements();
    ProviderPlugin plugin;
    if( plugins.hasMoreElements() )
    {
        plugin = (ProviderPlugin)plugins.nextElement();
        plugin.updateProviderCapabilities( capabilities );
    }
    while( plugins.hasMoreElements() )
    {
        plugin = (ProviderPlugin)plugins.nextElement();
        plugin.updateProviderCapabilities( tmpCapabilities );
        capabilities.and( tmpCapabilities );
        tmpCapabilities.reset();
```

```
        }
    }

    public GenAddress getGenAddress( String addressName )
                                                    throws InvalidArgumentException
    {
        addressName = m_jtsmiPlugin.addressConvert( addressName );
        GenAddress address = getAddress( addressName );
        ProviderPlugin plugin;
        Boolean state;
        if( null == address )
        {
            String providerPluginName = m_jtsmiPlugin.pluginForAddress( addressName
);
            plugin = (ProviderPlugin)m_plugins.get( providerPluginName );
            if( plugin == null )
            {
                throw new PlatformException(
                        "MultyPluginJTSMI.getGenAddress: invalid plugin name ");
            }
            state = (Boolean)m_pluginStates.get( plugin);
            if( !state.booleanValue() )
            {
                throw new PlatformException(" The provider plugin is out of
service");
            }
            address = m_creator.createRemoteAddress(m_provider, plugin, addressName);
        }
        else
        {
            plugin = address.getProviderPlugin();
            state = (Boolean)m_pluginStates.get( plugin);
            if( !state.booleanValue() )
            {
                throw new PlatformException(" The provider plugin is out of
service");
            }
        }
        return address;
    }

    public GenAddress getAddress( String addressName )
                                                    throws InvalidArgumentException
    {
      addressName = m_jtsmiPlugin.addressConvert( addressName );
      if( addressName == null )
      {
                throw new InvalidArgumentException( "null address number" );
        }
        return (GenAddress)m_addresses.get( addressName );
    }

    public GenTerminal getGenTerminal( String terminalName )
                                                      throws InvalidArgumentException
    {

        GenTerminal terminal = getTerminal( terminalName );
       /* if( null == terminal )
        {
```

```
            terminal = m_creator.createRemoteTerminal(m_provider, m_plugin,
terminalName);
        } */
        return terminal;
    }

    public synchronized void inService(boolean state, ProviderPlugin plugin)
                        throws IllegalStateException, IllegalArgumentException
{
    if( null == plugin )
    {
        throw new IllegalArgumentException( " null plugin object " );
    }
    Boolean newState = ( state ) ? ms_inService : ms_outOfService;
    Boolean oldState = (Boolean)m_pluginStates.get(plugin);
    if( !newState.equals(oldState) )
    {
        m_pluginStates.put( plugin, newState );
        changeProviderState();
    }
    if( ! state )
    {
        try
        {
            Call[] calls = m_provider.getCalls();
            if( calls != null)
            {
                for( int i=0; i<calls.length; i++)
                {
                    GenConnection[] conn =
                                (GenConnection[])calls[i].getConnections();
                    for( int j=0; j< conn.length; j++)
                    {
                        if( conn[j].getProviderPlugin().equals(plugin))
                        {
                            conn[j].disconnect();
                        }
                    }
                }
            }
        }
        catch ( Exception e)
        {
            GenJtapiPeer.ms_log.errorMessage(
                                "MultyPluginJTSMI.inService " , e);
        }
    }
}

private void changeProviderState()
{
    Hashtable table = new Hashtable( m_plugins.size() );
    Enumeration plugNames = m_plugins.keys();
    String pluginName;
    while( plugNames.hasMoreElements() )
    {
        pluginName = (String)plugNames.nextElement();
        table.put( pluginName,
                    m_pluginStates.get( m_plugins.get(pluginName)));
```

```
        }
        int state = m_provider.getState();
        // we cannot change the SHUTDOWN state of provider.
        if( m_jtsmiPlugin.isInService( table ))
        {
            if( state == Provider.OUT_OF_SERVICE )
            {
                m_provider.moveToState( Provider.IN_SERVICE );
            }
        }
        else
        {
            if( state == Provider.IN_SERVICE )
            {
                m_provider.moveToState( Provider.OUT_OF_SERVICE );
            }
        }
    }

    public boolean isPluginInService( ProviderPlugin plugin)
    {
        Boolean state = (Boolean)m_pluginStates.get(plugin);
        if( state != null )
        {
            return state.booleanValue();
        }
        return false;
    }
}
```

```
package com.ibm.hrl.jtapi.jtspi;

/*
 * (c) Copyright IBM Corporation 1998,1999,2000
 * IBM Research Laboratory in Haifa
 * Generic JTAPI Implementation (JTAPI 1.3)
 * ----------------------------------------
 * Package   : com.ibm.hrl.jtapi
 * Interface : HybridProviderPlugin
 * Created   : 13/06/2000
 */

public interface HybridProviderPlugin
{

    // IBM Copyright
    public static final String IBM_Copyright = Copyright.SHORT_STRING;


    /**
     * Informs the plugin that a given destination (remote) party has
     * answered the call.
     *
     * @param callId The ID of the given <code>Call</code> object.
     * @param remoteAddr The given remote <code>Address</code> object name.
     * @param remoteTerm The given remote <code>Terminal</code> object name.
     *
     * @exception IllegalArgumentException if a <code>null</code> argument
     *                                     is passed.
     * @exception IllegalStateException if the current state of an object
     *                                  involved in this method doesn't
     *                                  meet the acceptable conditions.
     * @exception JtspiException if a platform-specific exception occurred.
     */
     public void prvConnected (String callId, String remoteAdar, String remoteTerm)
                    throws IllegalArgumentException, IllegalStateException,
                           JtspiException;

     /**
      * Informs the plugin that a remote party has disconnected from a
      * <code>Call</code> associated with the given callId.
      *
      * @param callId The ID of the given <b>Call</b> object.
      * @param remoteAddr The given remote <b>Address</b> object name.
      * @param remoteTerm The given remote <b>Terminal</b> object name.
      *
      * @exception IllegalArgumentException if a <code>null</code> argument
      *                                     is passed.
      * @exception IllegalStateException if the current state of an object
      *                                  involved in this method doesn't
      *                                  meet the acceptable conditions.
      * @exception JtspiException if a platform-specific exception occurred.
      */
      public void prvDisconnected(String callId, String remoteAddr, String remoteTerm)
                    throws IllegalArgumentException, IllegalStateException,
                           JtspiException;

    /**
     * Tells the Generic JTAPI that the connection associated with the
```

```
     * given callId and the party's objects has failed.
     *
     * @param callId The ID of the given <b>Call</b> object.
     * @param partyAddr The given party <b>Address</b> object name.
     * @param partyTerm The given party <b>Terminal</b> object name.
     * @param cause The cause of the failure.
     *
     * @exception IllegalArgumentException if a <code>null</code> argument
     *                                     is passed.
     * @exception IllegalStateException if the current state of an object
     *                                     involved in this method doesn't
     *                                     meet the acceptable conditions.
     * @exception JtspiException if a platform-specific exception occurred.
     */
    public void prvFailed(String callId, String partyAddr,
                          String partyTerm, int cause)
                   throws IllegalArgumentException, IllegalStateException,
                          JtspiException;

    /**
     * Tells the Generic JTAPI that the connection is ringing at the
     * remote party's terminal.
     *
     * @param callId The ID of the given <b>Call</b> object.
     * @param remoteAddr The given remote <b>Address</b> object name.
     * @param remoteTerm The given remote <b>Terminal</b> object name.
     *
     * @exception IllegalArgumentException if a <code>null</code> argument
     *                                     is passed.
     * @exception IllegalStateException if the current state of an object
     *                                     involved in this method doesn't
     *                                     meet the acceptable conditions.
     * @exception JtspiException if a platform-specific exception occurred.
     */
    public void prvRinging (String callId, String remoteAddr, String remoteTerm)
                   throws IllegalArgumentException, IllegalStateException,
                          JtspiException;

    /**
     * Tells the Generic JTAPI that there is a new party was added to the
     * <code>Call</code>.
     *
     * @param callId     The ID of the given <code>Call</code> object.
     * @param newAddr    The <code>Address</code> object name of the new party.
     * @param newTerm    The <code>Terminal</code> object name of the new party.
     * @param plugin     The reference to the calling provider plugin.
     *
     * @exception IllegalArgumentException if a <code>null</code> argument
     *                                     is passed.
     * @exception IllegalStateException if the current state of an object
     *                                     involved in this method doesn't
     *                                     meet the acceptable conditions.
     * @exception JtspiException if a platform-specific exception occurred.
     */
    public void prvPartyAdded(String callId, String newAddr, String newTerm )
                   throws IllegalArgumentException, IllegalStateException,
                          JtspiException;
```

```
/**
 * Tells the Generic JTAPI that the two <code>Call</code> objects were
 * merged.
 * <p>
 * If the both <code>Call</code> objects belong to this
 * Generic JTAPI, the Generic JTAPI merges the corresponding objects and
 * the method returns <code>true</code>.
 * <p>
 * If only call with first ID belong to this Generic JTAPI, the method
 * returns <code>false</code>, and the Generic JTAPI will expect to the
 * states updating by <code>partyAdded</code> method.
 * <p>
 * If only call with second ID belong to this Generic JTAPI, the ID of the
 * call will be changed, the method returns <code>false</code> and the
 * Generic JTAPI will expect to the states updating by
 * <code>partyAdded</code> method.
 * <p>
 * If the both calls are not associated with this Generic JTAPI, nothing
 * will be done and the method returns <code>false</code>
 * <p>
 * @param targetId The Id of the stayed <code>Call</code> object.
 * @param sourceId The name of the swallowed up <code>Call</code> object.
 *
 * @return <code>true</code> if the state changing was finished.
 * @exception IllegalArgumentException if a <code>null</code> argument
 *                                     is passed.
 * @exception IllegalStateException if the associated <code>Call</code>
 *                                  objects is not in the ACTIVE state.
 * @exception JtspiException if a platform-specific exception occurred.
 */
public boolean prvCallsMerged(String targetId, String sourceId)
                  throws IllegalArgumentException, IllegalStateException,
                         JtspiException;

/**
 * Tells the Generic JTAPI that the <code>Connection</code> associated with
 * a given CallId and <code>Address</code> was redirected to a new
 * destination <code>Address</code>.
 *
 * @param callId  The given <code>Call</code> object ID.
 * @param oldAddr The name of the current <code>Address</code> object.
 * @param newAddr The name of the given new destination
 *                <code>Address</code> object.
 *
 * @exception IllegalArgumentException if a <code>null</code> argument
 *                                  is passed or there is no association
 *                                  between given call party objects.
 * @exception IllegalStateException if the current state of an object
 *                                  involved in this method doesn't meet
 *                                  the acceptable conditions.
 * @exception JtspiException if a platform-specific exception occurred.
 *
 */
public void prvRedirected(String callId, String oldAddr,
                          String newAddr)
                throws IllegalArgumentException, IllegalStateException,
                       JtspiException;

/**
```

```
 * Tells the Generic JTAPI that the <code>Call</code> object was transfered
 * <p>
 *
 * @param callId The Id of the <code>Call</code> object.
 * @param address  The name of the <code>Address</code> associated with the
 *                 transfer controller <code>TerminalConnection</code>.
 * @param terminal The name of the <code>Terminal</code> associated with the
 *                 transfer controller <code>TerminalConnection</code>.
 * @param newAddr  The name of the given new destination
 *                 <code>Address</code> object.
 *
 * @exception IllegalArgumentException if a <code>null</code> argument
 *                                     is passed.
 * @exception IllegalStateException if the associated <code>Call</code>
 *                                  objects is not in the ACTIVE state.
 * @exception JtspiException if a platform-specific exception occurred.
 */
public void prvTransfered(String callId, String address,
                          String terminal, String newAddress)
               throws IllegalArgumentException, IllegalStateException,
                      JtspiException;

/**
 * Tells the <code>Provider</code> object that one or more DTMF digits
 * have been detected.
 *
 * @param _callId   The ID of the given <b>Call</code> object.
 * @param _address  The name of the given <code>Address</code> object.
 * @param _terminal The name of the given <code>Terminal</code> object.
 * @param _digits   The Dtmf digits that were detected.
 *
 * @exception IllegalArgumentException if a <code>null</code> argument is
 *                                     passed.
 * @exception IllegalStateException if the current state of an object
 *                                  involved in this method doesn't meet the
 *                                  acceptable conditions.
 * @exception JtspiException if a platform-specific exception occurred.
 */

public void prvDtmfDetected(String _callId, String _address, String _terminal,
                            String _digits)
               throws IllegalArgumentException, IllegalStateException,
                      JtspiException;

/**
 * Tells the <code>Provider</code> object that the availability of the
 * media for a <code>TerminalConnection</code> has changed.
 *
 * @param _callId    The ID of the given <b>Call</code> object.
 * @param _address   The name of the given <code>Address</code> object.
 * @param _terminal  The name of the given <code>Terminal</code> object.
 * @param _available <code>true</code> if media is available, otherwise -
 *                   <code>false</code>.
 *
 * @exception IllegalArgumentException if a <code>null</code> argument is
 *                                     passed.
 * @exception IllegalStateException if the current state of an object
 *                                  involved in this method doesn't meet the
 *                                  acceptable conditions.
```

```
    * @exception JtspiException if a platform-specific exception occurred.
    */
    public void prvSetMediaAvailability(String _callId, String _address,
                                        String _terminal, boolean _available)
                    throws IllegalArgumentException, IllegalStateException,
                           JtspiException;

}
```